Detecting the Temporal Context of Queries

Oliver Kennedy[1], Ying Yang[1], Jan Chomicki[1],
Ronny Fehling[2], Zhen Hua Liu[2], and Dieter Gawlick[2]

[1] University at Buffalo, SUNY, Buffalo, NY 14260, USA,
`{okennedy,yyang25,chomicki}@buffalo.edu`
[2] Oracle Corp., Redwood Shores, CA 94065, USA,
`{ronny.fehling,zhen.liu,dieter.gawlick}@oracle.com`

Abstract. Business intelligence and reporting tools rely on a database that accurately mirrors the state of the world. Yet, even if the schema and queries are constructed in exacting detail, assumptions about the data made during extraction, transformation, and schema and query creation of the reporting database may be (accidentally) ignored by end users, or may change as the database evolves over time. As these assumptions are typically implicit (e.g., assuming that a sales record relation is append-only), it can be hard to even *detect* that a mistaken assumption has been made. In this paper, we argue that such errors are consequences of unintended *contextual dependence*, i.e., query outputs dependent on a variable characteristic of the database. We characterize contextual dependence, and explore several strategies for efficiently detecting and quantifying the effects of contextual dependence on query outputs. We present and evaluate our findings in the context of a concrete case study: Detecting temporal dependence using a database management system with versioning capabilities.

1 Introduction

The goal of analytical modeling is to reflect as accurately as possible the real world. However, with data often being machine-generated, people are increasingly realizing that "a single version of the truth ceases to be absolute and becomes relative and contextual."[1] Being able to relate the results of a query to the contextual assumptions on which the base data is built is thus increasingly critical for providing accurate and timely results.

In this paper, we explore the notion of data context: First, we identify *contextual dependence* as a relationship between query outputs and *implicit* contextual filters on the data being used. Then, we address a concrete class of contextual dependence: *temporal dependence*, which occurs when a query result depends on data that changes over time. Finally, we survey and evaluate multiple strategies for what we call *dependence analysis*: the detection of contextual dependence in query results.

[1] `http://blogs.forrester.com/boris_evelson/12-12-12-top_10_bi_predictions_for_2013_and_beyond`

1.1 Motivation

The root challenge of contextual analysis arises across a variety of domains. We will discuss three specific domains that directly benefit from contextual analysis.

Contextual Dependence Errors in ETL Processes. Extract, Tranform, Load (ETL) is the process by which data is typically introduced into a data management system. Because of the nature of ETL, contextual assumptions about or dependencies in a data set or its structure can be omitted or misrepresented. ETL processes typically mask the source data and the transformation — only the final result is loaded into the analytical warehouse. For example when an ETL process aggregates data, individual values are lost and an outlier might skew the results in a misleading way. As a consequence, end-users who are not aware of such masked properties of the original data may inadvertently issue queries that mischaracterize it.

Example 1. Consider an ETL process that periodically dumps an OLTP customer database to an OLAP database for analysis. At the moment the ETL process runs, Alice's credit account balance might be $10,000. This simplified representation is natural, but might be misleading for analysts — for example, Alice's typical credit balance of $0 might be masked by a recent purchase.

Dependence analysis is a first step towards ensuring user awareness of masked properties (e.g., by visually annotating query results). Identifying tuples and attributes that change with variations in the ETL process can serve as a quick mental sanity check for end-users, to ensure that these variations are expected and/or within acceptable tolerances. Note that this issue could be addressed by a carefully designed ETL process *if* queries are known apriori, or handled aposteriori in query logic *if* this dependence can be identified or tracked. Context analysis *eases the burden* of detecting dependence in the first place.

Optimizing Operational Business Intelligence. Operational business intelligence, sometimes referred to as real-time business intelligence uses the analysis of real-time or low latency data to enable a continual view into what is happening, and to support decision making for a company. While integration, processing and analysis of near-real-time information is needed across the board, it can require significant cost in software, hardware and staff to develop and deploy [9].

We believe that a *fully* real-time decision query support system might not always be the best answer for operational BI. For example, consider a threshold on the inventory of certain items. In order to decide whether new items need to be ordered, there is no real need to constantly evaluate the exact number of items currently on the shelves until an inflection point is crossed which marks the boundary of inventory. In the past, such thresholds have been determined manually and were typically static. Predicting the optimal thresholds based on live sales data has resulted in higher efficiency and profits. However, creating these models can be very challenging and costly. Contextual analysis on the time domain could suggest models, again easing the burden on end users.

Volatility Analysis. Similarly, by analyzing the historic volatility for certain columns, a system could autonomously provide insight into another big BI data quality problem: missing data. For example, if a column historically has a certain update rate, and suddenly that rate changes outside of the norm, the system can notify the user accordingly.

In this paper, we characterize the notion of contextual analysis using time as a context in Sections 3 and 4. In Section 5 we survey and adapt multiple strategies, both exact and approximate, for detecting and quantifying the extent of temporal dependence in a query result. We implement these strategies on a commercial database system and present the result of our evaluation of their performance and accuracy in Section 6.

2 Related Work

There has been much work on the use of database versioning to track the evolution of data over time [7, 8, 18, 21], and on providing low latency access to temporal and versioned data using temporal indexing methods [22, 16]. Many existing production-grade databases support SQL 2011's bi-temporal features [15]; This and related research efforts are basic technology enablers for our own work. Our primary focus is on performing query analysis *in-situ over existing database applications* by using transaction-time support already in the database. We evaluate and compare a range of strategies based on Monte-Carlo sampling, temporal queries, and provenance.

Our first approach uses Monte-Carlo sampling [11], a technique used in approximate query answering [19], and for probabilistic data [13]. We employ similar strategies to estimate the evolution of a query result over time.

We also explore the use of temporal database techniques [17, 5, 24] to optimize query evaluation. Point-in-time semantics allow non time-aware queries to be posed over temporal data, but do not admit an efficiently queryable data representation. Techniques for compiling point-in-time queries into interval-semantics queries [17] are a potential strategy for us, as they allow non-temporal queries to be evaluated efficiently over a version history.

Finally, we consider CTables and related work on provenance and incomplete information [12, 26, 4]. These strategies modify query execution to be aware of annotations tracking provenance [26, 4] or missing information [12]. We employ a simplified form of these techniques to connect pre-computed summaries of data variability in query inputs with the corresponding query outputs.

3 Data Context

In this paper we adapt the model of context proposed by Bertossi et. al. [2], and treat the relations of a database as views over a "global" database instance extended with additional contextual metadata. The Bertossi model classifies data as being of either high or low quality using a set of predicates over both the database and its surrounding context. Different quality metrics are achieved by varying the set of predicates considered. Rather than using a binary classifier, we instead consider context as establishing an *enumerable* space of possible interpretations.

The space of possible interpretations is captured by a *singleton* relation C with attributes encoding factors *implicitly* assumed by a database developer and/or the user posing a query. For example, the context relation might consist of a user's current location, a database designer's choice of Fahrenheit or Celsius, and/or a vector of exchange rates over time. Let G be a global database instance with schema S_G. We define a context-aware database instance D_C as a database instance with schema S_G and relations $R_C \in D_C$ defined as views over G:

$$R_C(x) \Leftarrow \phi_G(x) \wedge C(x)$$

Here $\phi_G(x)$ is an arbitrary conjunctive query over G. We refer to the relation C as the *context* of D_C. D_C captures the effects of context C under the set of assumptions that form C.

Given a query $Q(D_C)$ and a set $\mathcal{C}$ of contexts (i.e., instances of C), the goal of context analysis is to determine if and how $Q(D_C)$ depends on C. We refer to such dependencies as table-, tuple-, or attribute-level *dependence* in Q, depending on what parts of $Q(D_C)$ are affected.

Example 2. An analyst is working with a database G which includes temperature readings, and poses a query $Q(G)$. Using units of temperature as an assumption, there might be two possible contexts: one in which G's temperatures are in Celsius, and another in which they are in Fahrenheit. This gives us two instances D_c, D_f, respectively. The query $Q(G)$ can now be restated as a choice between the results of either $Q(D_c)$ or of $Q(D_f)$, depending on which contextual assumption is true. If $Q(D_c) \equiv Q(D_f)$, we say that Q does not have a contextual dependence. Similarly, a tuple present only one of $Q(D_c)$ and $Q(D_f)$ exhibits a tuple-level context dependence.

We are not only interested in the presence of context dependence, but in quantifying its *impact*. Following the literature on representing and querying incomplete information [12, 1, 13, 14], one form of impact is to measure the subset of $\mathcal{C}$ for which a given tuple is present in $Q(D_C)$ (the tuple's *confidence*). Another is to generate visualizations and analytical summaries (moments, boundaries, or trends) for individual attribute values in the output. Such analyses generally take the form of a query aggregating over all possible contexts. This common, simple structure admits a range of different evaluation strategies. Our goal in this paper is to enumerate and evaluate four such strategies: (1) A naive, evaluation for each context, (2) a monte-carlo based sampling strategy, (3) a strategy based on query rewriting, and (4) an approximation strategy that composes precomputed partial analyses.

Time as Context. A particularly important instance of contextual dependence occurs when a relation's context changes over time. This class of dependence is frequently opaque to users, making it extremely hard for humans to detect. Moreover, many commercial databases already have transaction time and/or versioning capabilities and natively provide support for views of data parameterized by time. This makes the detection of contextual dependences feasible to implement, *even for databases already deployed in production.*

Example 3. Returning to Example 1, an analyst decides to run a new promotion for customers with low balances. Without realizing the volatility of the balance attribute, the analyst queries for all customers with a balance under \$1000 and mistakenly excludes Alice due to her recent purchase. The OLTP database's version history can be used to correctly answer the analyst's query, only after the analyst comes to realize that an error has occurred. Temporal dependence analysis can serve as a quick sanity check for the analyst, allowing her to quickly discover the potential error.

To define the view R_C, we take a context C consisting of a single attribute: *time* chosen from an enumerable *time horizon of interest.* The view R_t defines the state of relation R at the *point in time t*. We refer to a contextual dependence where time is the primary feature of the context as a *temporal dependence.* For the remainder of this paper we will focus on detecting temporal dependence in queries.

4 Detecting Temporal Dependence

We consider three levels of detail when searching for temporal dependence in query results: (1) Detecting temporal dependence in the entire result relation: *Does the set of customers with balances under \$1000 change over time?*, (2) Detecting temporal dependence in individual result attributes: *Does Alice's balance change over time?*, and (3) Quantifying the effects of temporal dependence on the results: *How much does Alice's balance change over time?*.

Naive Temporal Dependence Analysis. As a baseline, we consider a naive strategy based on a process called *temporal query normalization* [17, 5, 24]. Normalization identifies a minimal set of time intervals over which the results of a query Q are guaranteed to be unchanged. This is accomplished by enumerating the full list of versions at which a change is applied to any base relation referenced by Q ($versions(Q)$ for short). The query is evaluated once for each version in $versions(Q)$, producing a set of result relations (a set of sets) $R_Q = \{Q(D_t) \mid t \in versions(Q)\}$. Clearly, if $|R_Q| > 1$ then Q has a temporal dependence.

This approach presents two immediate challenges: First, detecting (and quantifying) row-level changes with respect to time requires matching corresponding rows between relations in R_Q. Second, this approach can require substantial wasteful re-evaluation of large portions of the query, especially if each version makes only minor changes to the base relation. We first consider the question of row matching, and return to evaluation strategies in Section 5.

4.1 Row Matching

We would like to be able to identify not only row-level differences (i.e., insertions and deletions), but also cell-level differences (i.e., updates to individual data values). To accomplish this, it is necessary to match rows across query evaluations on different versions of the database — Was a row of the output modified (i.e., a cell-level change) or replaced in its entirety (i.e., a row-level change)?

We would like to match query output rows according to conceptual equivalence if at all possible, pairing two outputs if they correspond to the same conceptual entity. We consider three classes of queries for which it is possible to exactly detect conceptual equivalence:

Aggregate Queries The group-by attributes of an aggregate query act as a unique identifier, or key for each output tuple.

Distinct Queries The distinct attributes of a query act as a key.

Single-Relation Queries The key (if available) of the relation being queried is used to identify each output tuple. For base relations that do not have keys (i.e., bags), we use the relation's physical row identifier attribute as a key.

For queries that do not fall into one of the above categories, we approximate conceptual equivalence through a form of why-provenance [6, 3]. To support non-aggregate join queries, we compute a new key for each query output by composing the keys of each input source. The projection targets of the query are then extended with the key. Throughout the rest of this paper, we will assume that all queries have been transformed according to this process, and that the key is known.

4.2 Impact Assessment and Visualization

A copy of the query is evaluated over each version in a user-specified time horizon of interest: $t_1 \ldots t_N$. We abuse notation and treat t as the entire contextual interpretation C, and define AS-OF [18] queries as $Q^t(D) := Q(D^t)$ Visual annotations such as colored highlights (or asterisks in a purely text-based setting) on the query output convey two pieces of information to users:

1. Rows present in the current output are marked if they were not present in the output at some point in the time horizon of interest. The entire output table is marked if rows in the table were removed at some point in the time horizon of interest.
2. Individual (non-key) cells are marked if their values have changed at some point in the past.

In addition to highlighting temporal dependence in results, we wish to present the user with a significance metric for each case of temporal dependence. This is accomplished by grouping tuples together by key with an aggregate function of the form:

$$\texttt{SELECT } key\texttt{, AGG(...) FROM } Q^1 \texttt{ UNION ... UNION } Q^N \texttt{ GROUP BY } key$$

We focus on two aggregate metrics in this paper: row confidence, and attribute boundaries. More general aggregate summaries like average, standard deviation, and even plots over time can also serve as useful metrics or visualizations of the significance of a specific temporal dependence.

Confidence. We use confidence to capture the significance of a row-level temporal dependence. The confidence of a given row r is the fraction of time during which it is present in the database. This may be defined in terms of versions (e.g., as the percent of versions during which the row is present in the output), or by weighting (e.g., by wall-clock time):

```
SELECT key, COUNT(*) / N FROM Q^1 UNION ... UNION Q^N GROUP BY key
```

Boundaries. Boundaries or ranges capture the significance of an attribute-level temporal dependence. The upper and lower bounds for each attribute are computed by matching key attribute values across all results.

```
SELECT key, MakeRange(MIN(A_1), MAX(A_1)) AS A_1, ...
FROM Q^1 UNION ... UNION Q^N GROUP BY key
```

Example 4. Returning to Example 3, we would like to make it possible for the OLAP interface to signal the analyst in one of three ways: (1) By marking the analyst's balance query as time-dependent, (2) By marking Alice's presence in the result set as time-dependent, and/or (3) By indicating that Alice is part of the result set 99% of the time.

5 Practical Temporal Dependence Detection

This naive strategy requires many complete evaluations of the query. We now survey a variety of specialized query evaluation techniques designed for related scenarios, and discuss how they may be applied to support efficient detection of temporal dependence in results. We consider three strategies: approximation algorithms including *Monte-Carlo Approximation*, and one we call *Analysis Composition*, as well as an exact algorithm that we call *Dynamic Analysis.*

5.1 Monte-Carlo Approximation

Our first approach attempts to limit the amount of work performed during temporal dependence detection by using Monte-Carlo approximation. Instead of evaluating the query at every version, the query is only evaluated on a fixed set of randomly selected sample versions S. In other words:

$$\left(\bigcup_{1\leq t\leq N} Q^t\right) \approx Q^S := \left(\bigcup_{t\in S} Q^t\right)$$

Statistical metrics for the result set are computed as in the naive approach, but on a smaller set of sample points. Confidence values, expectations, and range boundaries are independent of result-set size; The process for computing these metrics is entirely unchanged. The time complexity of detecting temporal dependence in Q^S scales linearly in $|S|$, which we can control directly, rather than in N, which we can not. The improved performance comes at the cost of result accuracy.

5.2 Dynamic Analysis

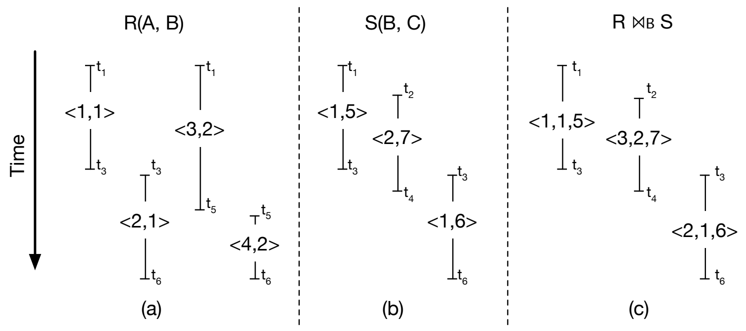


Fig. 1. An example dataset illustrating the value of dynamic analysis. The histories of relations $R(A, B)$ and $S(B, C)$ contain (respectively) 4 and 3 tuples over the interval (t_1, t_6). The join of both relations is shown under dynamic analysis. Under static analysis, tuple $\langle 2, 1, 6\rangle$ would be partitioned into 3 intervals: (t_3, t_4), (t_4, t_5), and (t_5, t_6).

The normalization process described previously operates at a coarse granularity. A separate instance of the query is executed for every version where *any* output changes. Instead of discarding query instances to produce an approximation, computation can be shared across successive versions.

Consider the temporal database shown in Figure 1. Naively, evaluating $R \bowtie S$ completely for each version where R or S changes (i.e., for each of (t_1, t_2), (t_2, t_3), $\ldots$, (t_5, t_6)) involves redundant computation. For example, $\langle 1, 1\rangle \in R$ must be joined with tuple $\langle 1, 5\rangle \in S$ twice, once for the interval (t_1, t_2), and again for the interval (t_2, t_3), even though neither input tuple changes at version t_2.

In this section, we adapt temporal query compilation techniques developed by Lessa [17] to our substantially simpler setting, and present an optimized form of temporal query normalization that we call *dynamic analysis*. Instead of partitioning tuple intervals prior to query evaluation, intervals are partitioned on-demand, as needed. Because tuples are partitioned independently, the number of partitions created is much smaller.

Each tuple is annotated with a new pair of *interval* attributes *start* and *end*, denoting the first and last version during which the tuple is present in the database. Queries are rewritten to properly map the interval attributes of their input relations to their output tuples.

Select, Project. Selections do not interact with intervals, and remain unchanged. Projections are modified to preserve the interval annotating each input tuple. For example, the query `SELECT A FROM R` would be rewritten to preserve the interval annotation: `SELECT A, start, end FROM R`

Join. Joins normalize output tuples on a tuple-by-tuple basis. The joined tuple is only present in the database so long as both tuples that generated it are present; The output tuple's interval is the intersection of the intervals of its sources. If this interval is empty, the tuple can be dropped immediately. For example, the query `SELECT A FROM R, S WHERE R.B = S.B` is rewritten as:

```
SELECT A, GREATEST(R.start,S.start) AS start,
          LEAST(R.end,S.end) AS end
FROM R, S WHERE R.B = S.B
          AND GREATEST(R.start,S.start) < LEAST(R.end,S.end)
```

This process generalizes to any number of relations and projection attributes.

Aggregate. Unlike joins, an aggregate requires its inputs to be normalized first. Consider a query of the form `SELECT A, SUM(B) AS B FROM R GROUP BY A` For each version in which `R` is modified, the aggregate value of the group containing the modified tuple also changes. We embed this transition into the aggregate query through a synthetic relation: `VSET(A, B, vers)`, which includes every version where $\pi_{A,B}(\mathtt{R})$ changes. We compute the aggregate value for each interval, using the versions in `VSET` to mark the `end` of each generated interval.

```
SELECT R.A, V.vers, SUM(B) AS B FROM R, VSET V
WHERE V.A = R.A AND vers BETWEEN R.start AND R.end GROUP BY A
```

Each interval is now annotated with only the end-point. We construct a simple window query to obtain the matching start points for each interval. In the following query, the `NTH_VALUE` aggregate is used to obtain the endpoint of the previous interval for group A.

```
SELECT A, B, NTH_VALUE(vers, 1) OVER
      (PARTITION BY A ORDER BY vers ROWS 1 PRECEDING) AS start,
      V.vers AS end FROM QPOINT
```

This process generalizes trivially to multiple group-by attributes, or multiple aggregate values. If the aggregate query involves multiple source relations, the join component of the query is first isolated in a separate nested-from clause and then rewritten as a normal non-aggregate query.

Acct	Name	Balance	ROWID	XID
	Alice	100	r1	v1
	Bob	200	r2	v1
	Carol	5000	r3	v1

(a)

Acct	Name	Balance	ROWID	XID
	Alice	10,000	r1	v2
	Carol	5000	r3	v1
	Carol	55	r4	v2

(b)

Fig. 2. The Table `Acct` extended with row id `ROWID` and transaction id `XID` metadata (a) before updates, and (b) after updates.

Example 5. Recall our running example, and consider the example Accounts relation `Acct` shown in Figure 2a. A sequence of updates is applied to this relation as follows, and the final relation is shown in Figure 2b.

```
UPDATE Acct SET Balance = Balance + 9900 WHERE Name = 'Alice';
INSERT INTO Acct VALUES ('Carol', 55); COMMIT; -- Creates v2
DELETE FROM Acct WHERE Name = 'Bob'; COMMIT;   -- Creates v3
```

Acct	Name	Balance	RID	start	end
	Alice	100	r1	$-\infty$	v2
	Alice	10,000	r1	v2	∞
	Bob	200	r2	$-\infty$	v2
	Carol	5000	r3	$-\infty$	∞
	Carol	55	r4	v2	∞

(a)

Q	Name	Balance	start	end
	Alice	100	$-\infty$	v2
	Alice	10,000	v2	∞
	Bob	200	$-\infty$	v2
	Carol	5000	$-\infty$	v2
	Carol	5055	v2	∞

(b)

Fig. 3. An interval encoding of the history of the relation `Acct` from Figure 2 and the result of query Q on `Acct`.

The analyst poses a query $Q :=$ `SELECT Name, SUM(Balance) FROM Acct GROUP BY Name`, and wishes to detect temporal dependence in the output. To begin, the internal representation of `Acct` uses an interval encoding as shown in Figure 3a. Following the dynamic analysis rules, Q is rewritten as follows.

```
SELECT Name, Balance, NTH_VALUE(pt, 1) OVER
         (PARTITION BY Name ORDER BY pt ROWS 1 PRECEDING) AS start,
       pt AS end
FROM (SELECT A.Name, V.pt, SUM(Balance) FROM Acct, (
                SELECT DISTINCT Name, start AS pt FROM Acct
          UNION SELECT DISTINCT Name, end   AS pt FROM Acct
        ) V WHERE V.A = R.A AND V.pt BETWEEN R.start AND R.end
        GROUP BY Acct.Name, V.pt
     ) QPOINT
```

Finally, the impact assessment process of Section 4.2 aggregates the result to obtain confidence and range values. `Name` is a key for Q, resulting in the following summary relation:

Q	Name	Balance	confidence
	Alice	[100, 10000]	1
	Bob	[200, 200]	0.75
	Carol	[5000, 5055]	1

Alice's balance is highly volatile, but her account is open for all four versions resulting in a confidence of 1. Bob's account balance is stable, but he closes his account in `v3`. Carol opened a new account in `v2`, but the query only returns the total balance for each customer; The small fluctuation in her balance reflects this.

5.3 Analysis Composition

Finally, we consider an approach inspired by the C-Tables [12] encoding for incomplete information. Instead of analyzing queries directly, this approach instead *composes* already precomputed results to approximate the result of a full analysis.

A C-Table encodes many possible instantiations of a base relation in a compact encoding using labeled nulls to represent missing attribute values, and tuple annotations to mark tuples that only exist in a subset of all instantiations. Although originally targeted at representing incomplete information, C-Table-style encodings have been applied to other forms of partial information, including probability distributions [1, 14, 20] and generalized provenance [10]. Exact probability computations for C-Table queries can be #P complete [1], so we instead use a simpler uncertainty model based on pessimistic error bounds.

Precomputation. For each base relation R, we precompute a summary relation $S(R)$, which contains the output of a full temporal dependence analysis of the trivial query `SELECT * FROM` R as described in prior sections. As an additional optimization before analyzing each base relation, we determine which columns are fixed, and which are time-dependent within the time horizon of interest by performing a `COUNT DISTINCT` grouping by the full relation schema.

Based on the sets of fixed and time-dependent columns, $S(R)$ tracks ranges for each time-dependent attribute and confidence values for each row. Concretely, the schema $sch(S(R))$ includes: (1) a confidence attribute, which we label $CONF$, (2) all of the fixed attributes in $sch(R)$, and (3) two attributes a_{min} and a_{max} for each time-dependent attribute $a \in sch(R)$ representing upper and lower bounds for the attribute, respectively.

This compact encoding discards the specific distribution of values that each time-dependent attribute can take, as well as any temporal correlations that might exist between attributes. When querying this data, we make two simplifying assumptions: First, we compensate for discarding correlations by computing only pessimistic, worst-case bounds on the ranges output for each attribute. Second, for confidence computations, we assume a uniform distribution for values in each range.

SPJ Query Rewriting. Consider a query $Q(R_1, \ldots, R_n)$ which we wish to analyze for temporal dependence. Given summary relations $S(R_1)$, ..., $S(R_n)$, we wish to construct a query $Q'(S(R_1), \ldots, S(R_n))$ that approximates the results of a full temporal dependence analysis on Q. We initially consider SPJ queries:

`SELECT` F_1, ..., F_j, D_1, ..., D_k `FROM` R_1, ..., R_n `WHERE` $\phi \wedge \psi$

where F and D are fixed and time-dependent attributes, respectively, and ϕ is a boolean formula over both types of attributes, while ψ is a boolean formula exclusively over fixed attributes. Adapting the rules for C-Tables query rewriting [12] to our simplified model, the following three transformations are applied. (1) Time-dependent attributes are replaced by their corresponding range attributes D_{min}, D_{max}, (2) The confidence of the output relation is computed as the product of confidences of the input relations, and (3) The expression ϕ does not produce a binary truth value, but rather a confidence value that is combined with confidence values from the input sources. The above query is rewritten as:

$$\begin{aligned}&\texttt{SELECT } F_1, \ldots, F_j, D_{1,min}, D_{1,max}, \ldots, D_{k,max},\\ &\qquad \texttt{COMPUTE\_CONF}(\phi) \times R_1\texttt{.CONF} \times \ldots \times R_n\texttt{.CONF}\\ &\texttt{FROM } R_1, \ldots, R_n \texttt{ WHERE } \psi\end{aligned}$$

Here, `COMPUTE_CONF`(ϕ) computes the probability of ϕ being satisfied, assuming that each D_i uniformly takes values in the range $[D_{i,min}, D_{i,max}]$.

SPJA Query Rewriting. Aggregation is handled differently based on the aggregate function. Non-group-by aggregates always have a single row as output, fixing their confidence at 1. The confidence of a group is the cumulative probability that at least one tuple belonging to the group exists. Aggregates over time-dependent attributes are replaced with an expressions that compute hard upper and lower bounds of each as follows:

- `MAX`(D): The pessimistic upper bound on the maximum aggregate is the maximum of all D_{max} in the group. A lower bound could be computed similarly, but we must also account for the possibility that tuples may be removed from the aggregated value if they do not have 100% confidence. The pessimistic lower bound is the lowest value of D_{min} contained in a tuple with 100% confidence, or $-\infty$ if no such tuple exists.
- `MIN`(D): The upper and lower bounds are computed as the inverse of `MAX`.
- `SUM`(D): The pessimistic upper (resp., lower) bound includes contributions from all positive (resp., negative) integers D_{max} (resp. D_{min}), as well as all negative (resp., positive) integers in tuples with a 100% confidence.
- `COUNT`($*$): The pessimistic upper and lower bounds are the total number of tuples, and the total number of tuples with 100% confidence respectively.
- `AVG`(D) is computed as $\frac{\texttt{SUM}(D)}{\texttt{COUNT}(D)}$

6 Experiments

System Configuration. Experiments were performed on a commercial database system[2] with temporal query support and our query-rewriting front-end. The commercial database system was deployed on 2x8-core 2.6 GHz Intel Xeon processor system with 32GB of RAM, RHEL 6.5, and 4 300GB 10kRPM HDDs in a RAID 5 configuration. The front-end was deployed on a 3.7 GHz Intel i7 with 16GB of RAM, OS X 10.9, and a 256 GB SDD, connected to the server through a 100 mb/s ethernet network.

Benchmark Workload. We based our evaluation on the TPC-H benchmark [25] workload and dataset generators. In order to simulate real-world temporal dependence in the data, we extended TPC-H's synthetic update generator with additional events as follows:

[2] The commercial database system remains anonymous due to its licensing agreement.

- **(a) Place Order** (44%): An `orders` row and its matching `lineitem` rows as generated by the TPC-H update generator are inserted into the database. The `customer` who placed the `orders` has their `acctbal` debited the price of the order, and the `availqty` fields of matching `part` records are also adjusted.
- **(b) Cancel Order** (9%): An `orders` row and its matching `lineitem` rows as selected by the TPC-H delete generator are deleted from the database.
- **(c) Pay Supplier** (9%): The `acctbal` field of a `supplier` row selected uniformly at random is set to 0.
- **(d) Customer Pays** (9%): The `acctbal` field of a `customer` row selected uniformly at random is set to 0.
- **(e) Buy Part** (9%): An *amount* is selected uniformly in the range $[1, 1000]$, and a `partsupp` row is selected uniformly at random. The `availqty`, and `supplier.acctbalance` fields are updated accordingly.
- **(f) Change Part Price** (9%): A `part` row is selected uniformly at random, and its `retailprice` field is multiplied by a value uniformly chosen in the range $[0.95, 1.1]$.
- **(g) MSRP Changes** (2%): A `part.brand` value is selected uniformly at random, and all matching `part` rows have their `retailprice` fields multiplied by a value uniformly chosen in the range $[0.95, 1.1]$.
- **(h) Supplier Price Change** (9%): A `supplier` row is selected uniformly at random, and all corresponding `partsupp` fields have their `supplycost` multiplied by a value uniformly chosen in the range $[0.95, 1.1]$.

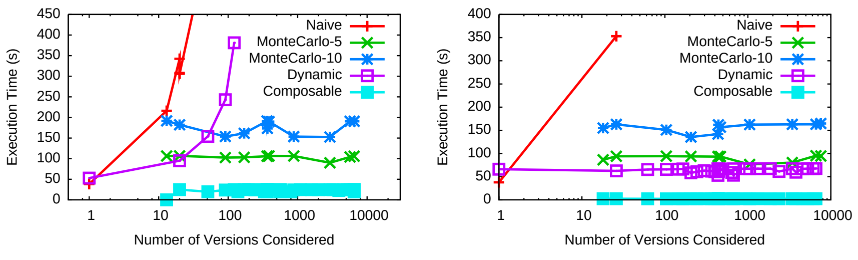
Experiments were run on a database initialized with a 1 GB TPC-H Dataset (Scaling Factor 1.0). The database's temporal features were then activated, and a sequence of approximately 100,000 events from the synthetic workload were applied to the database with a new version created after each event.

Algorithms. We consider the four temporal dependence detection strategies strategies presented in Sections 4 and 5. `Naive` denotes naive normalization, `MonteCarlo-X` denotes Monte Carlo sampling with X versions chosen uniformly at random, `Dynamic` denotes dynamic normalization, and `Composable` denotes composable analyses.

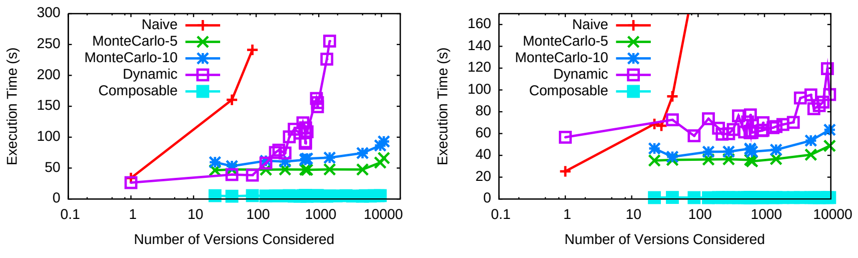
6.1 Performance Evaluation

Our experiments used a representative set of the TPC-H query workload including queries 1, 3, 9, and a lightly modified version of query 20. These queries cover a spread of functionality, requirements, and base relation update classes. Performance results are presented in Figure 4. We used each algorithm to analyze a single query for both confidence and attribute bounds, changing the time horizon of interest. For composable analysis, the pre-computation of each base-relation's analysis is not counted by our results, however even for the largest relation the process took less than two minutes. Results are presented in in Figure 4.

Query 1. is an aggregate query over a single table and consistently produces a set of four output rows. Without temporal query logic enabled, the query takes

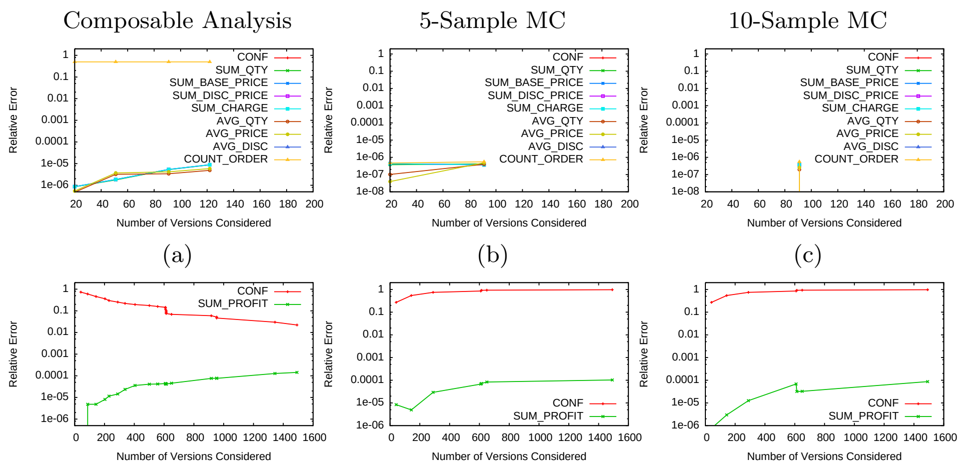


(a) TPC-H Query 1 (b) TPC-H Query 3



(c) TPC-H Query 9 (d) TPC-H Query 20

Fig. 4. Algorithm performance relative to size of the time horizon of interest. Note the log scale on the x axis.



(a) (b) (c)

(d) (e) (f)

Fig. 5. Accuracy of the approximate algorithms on TPC-H Queries 1 (a-c) and 9 (d-f). Each line represents the relative error introduced by the algorithm on a single attribute query output or the tuple confidence (`CONF`).

approximately 3 seconds to evaluate. As shown in Figure 4a, the approximate algorithms outperform the exact ones. Recall that for aggregates, dynamic analysis subdivides the versions created across all groups being output. Correspondingly, Dynamic analysis remains a constant factor (approximately 3–4 times) faster than the Naive strategy. Both approximate strategies show an expected near constant-time performance. The bulk of the work for the Monte Carlo approach is in generating a constant number of samples, while the size of each base relation summary remains roughly constant as the number of versions grows — note that the cost of generating the summaries is linear in the number of versions considered.

Query 3. is an aggregate query over a three-way hierarchical join. Without temporal query logic enabled, the query takes approximately 1.5 seconds to evaluate. As before, the approximate methods show linear scaling. However, Dynamic analysis also shows linear scaling. This is due to the hierarchical nature of the query. The query produces one output for each row of `orders` that satisfies the predicate. Update classes (a) and (b) insert and remove orders together with their corresponding `lineitem`, ensuring no attribute-level temporal dependence in the aggregate values.

Query 9. is an aggregate query over a six-way join grouping on a dimension attribute. Without temporal query logic enabled, this query takes approximately 3.5 seconds to compute. The number of groups is capped at the 25 nations in the TPC-H Schema, all of which are affected by updates in the base data. As before, Dynamic analysis is a constant factor faster than Naive analysis. We also note a slight linear increase in the cost of the Monte Carlo approach as the number of versions considered approaches 10,000. As this same pattern appears in our results for query 20 experiment, we suspect that this slight increase is due to the join width.

Query 20. is a non-aggregate query over multiple tables. We made two minor modifications to this query: (1) We manually applied a standard decorrelation rewrite [23], and used a `SELECT DISTINCT` query to emulate the `EXISTS` predicate. (2) We avoid non-numeric values in the query results, and compute the set of supplier account balances rather than the set of supplier addresses. Without temporal query logic enabled, this query takes approximately 5 seconds to compute.

6.2 Accuracy

Figure 5 shows a comparison of accuracy metrics for the approximation methods. Composable analysis is quite accurate, generating estimates to within four nines for most aggregate values. In Query 1, a significant error appears in the `COUNT_ORDER` attribute. This is a consequence of composable analysis' pessimistic lower bound estimate. Any row with a confidence value lower than 1 is dropped from the lower bound; The upper bound is accurate. For Query 9, the accuracy actually increases as more versions are considered: this is an artifact of the actual confidence getting closer to the pessimistic bound. The monte-carlo methods fared significantly better at estimating the aggregate values computations. However, they did not identify some output groups, creating an extremely high confidence error.

7 Conclusions

In this paper, we identified an important class of queries: Context analysis queries, which vary one or more contextual attributes to identify dependence on the context. We explored context analysis in the domain of versioned/temporal databases, and identified three strategies for leveraging temporal features in existing database systems to detect temporal dependence *in existing queries*: Sampling, Dynamic Analysis, and Composable Analysis. We evaluated these strategies on a commercial database system and found that Composable Analysis substantially out-performs the other strategies. This performance comes at the cost of accuracy, in particular when computing confidence values. As expected, Dynamic Analysis provided the best performance of the non-approximate strategies.

References

1. Lyublena Antova, Christoph Koch, and Dan Olteanu. {10ˆ{(10ˆ{6})}} worlds and beyond: efficient representation and processing of incomplete information. *VLDBJ*, 18(5):1021–1040, 2009.
2. Leopoldo Bertossi, Flavio Rizzolo, and Lei Jiang. Data quality is context dependent. In *BIRTE*, 2010.
3. Peter Buneman, Sanjeev Khanna, and Wang-Chiew Tan. Why and where: A characterization of data provenance. In *ICDT 2001*, 2001.
4. Peter Buneman and Wang-Chiew Tan. Provenance in databases. In *SIGMOD*, 2007.
5. Jan Chomicki and David Toman. Temporal databases. *Foundations of Artificial Intelligence*, 1:429–467, 2005.
6. Yingwei Cui, Jennifer Widom, and Janet L Wiener. Tracing the lineage of view data in a warehousing environment. *ACM TODS*, 25(2):179–227, 2000.
7. Peter Dadam, Vincent Y. Lum, and H.D. Werner. Integration of time versions into a relational database system. In *VLDB*, 1984.
8. K.R. Dittrich and R.A. Lorie. Version support for engineering database systems. *IEEE TOSE*, 14(4):429–437, Apr 1988.
9. Gartner. Predicts 2014: Business intelligence and analytics will remain cio's top technology priority. http://www.gartner.com/document/2629220.
10. Todd J. Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *PODS*, 2007.
11. W Keith Hastings. Monte carlo sampling methods using markov chains and their applications. *Biometrika*, 57(1):97–109, 1970.
12. Tomasz Imieliński and Witold Lipski Jr. Incomplete information in relational databases. *JACM*, 31(4):761–791, 1984.

13. Ravi Jampani, Fei Xu, Mingxi Wu, Luis Leopoldo Perez, Christopher Jermaine, and Peter J Haas. MCDB: a monte carlo approach to managing uncertain data. In *SIGMOD*, 2008.
14. Oliver Kennedy and Christoph Koch. PIP: A database system for great and small expectations. In *ICDE*, 2010.
15. Krishna Kulkarni and Jan-Eike Michels. Temporal features in SQL:2011. *SIGMOD Record*, 41(3):34–43, October 2012.
16. A. Kumar, V.J. Tsotras, and C. Faloutsos. Designing access methods for bitemporal databases. *IEEE TKDE*, 10(1):1–20, Jan 1998.
17. Demian Lessa. *Temporal model for program debugging and scalable visualizations*. PhD thesis, University at Buffalo, SUNY, 2013.
18. D. Lomet, R. Barga, M.F. Mokbel, and G. Shegalov. Transaction time support inside a database engine. In *ICDE*, April 2006.
19. Frank Olken. Efficient methods for calculating the success function of fixed-space replacement policies. Technical report, Lawrence Berkeley Lab., CA (USA), 1981.
20. Dan Olteanu, Jiewen Huang, and Christoph Koch. Sprout: Lazy vs. eager query plans for tuple-independent probabilistic databases. In *ICDE*, 2009.
21. Oracle. Oracle flashback. http://www.oracle.com/technetwork/database/features/availability/flashback-overview-082751.html.
22. Simonas Šaltenis and Christian S Jensen. R-tree based indexing of general spatio-temporal data. Technical Report TR-45, TimeCenter, 1999.
23. P. Seshadri, H. Pirahesh, and T.Y.C. Leung. Complex query decorrelation. In *ICDE*, Feb 1996.
24. David Toman. Point-based temporal extensions of sql and their efficient implementation. In *Temporal databases: research and practice*, pages 211–237. Springer, 1998.
25. Transaction Processing Performance Council. Tpc-h benchmark specification. http://www.tpc.org/tpch/.
26. Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical report, Stanford InfoLab, 2004.